

A RAND NOTE

# Beyond User Friendly

Robert H. Anderson, Norman Z. Shapiro

December 1989







RAND

The research described in this report was supported by
The RAND Corporation using its own research funds.



Published by The RAND Corporation
1700 Main Street, P.O. Box 2138, Santa Monica, CA 90406-2138

A RAND NOTE

N-2999-RC

# Beyond User Friendly

Robert H. Anderson, Norman Z. Shapiro

December 1989



RAND

## PREFACE

Information scientists at The RAND Corporation have had continuing interest in the design of effective interactive systems and in their impact on the productivity and structure of work groups and organizations. A series of RAND occasional papers released during the past decade has dealt primarily with the design and effects of electronic mail systems as one important example (Shapiro and Anderson, 1985; Eveland and Bikson, 1987; Anderson et al., 1989).

In studying the design of user-computer interfaces, one is struck by the conceptual confusion that abounds in the literature. This Note, first published in *The EDUCOM Review*, Vol. 24, No. 3, Fall 1989, explores this confusion and formulates a multi-dimensional scale with which to appraise such interfaces.

The Note should be of interest to both users and designers of interactive information systems.

## SUMMARY

What is meant when the interface to a computer software program is called "user-friendly"? In this Note, we describe six "easy to . . ." dimensions for user-computer interfaces that distinguish among aspects of a program's behavior often confused within the "user-friendly" rubric. Some of these dimensions are mutually antagonistic, so priorities and choices must be made depending on the user's context. We apply the six dimensions to aid in understanding the love/hate relationship some software interfaces engender. Through categorizations of user interface traits, such as the example proposed here, we hope to aid both the design and procurement of effective user-computer interfaces for software products.

## CONTENTS

PREFACE ..... iii

SUMMARY ..... v

Section
I. INTRODUCTION ..... 1

II. THE DIMENSIONS OF "EASY TO ..." ..... 3
1. Easy to Use ..... 3
2. Easy to Learn (and Teach) ..... 3
3. Easy to Relearn ..... 4
4. Easy to Unlearn ..... 4
5. Easy to Avoid Harm ..... 4
6. Easy to Support ..... 5

III. THE CONTEXT: PEOPLE, TASKS ..... 7
The Love/Hate Relationship ..... 8

IV. CONCLUSION ..... 10

REFERENCES ..... 11

## I. INTRODUCTION

We believe that the buzzword "user-friendly" has become so lacking in content and specificity as to be virtually meaningless. Figure 1 shows an extreme illustration of this.[1]

What is meant by "user-friendly," focusing our attention on the *interface* to a *computer software* program? Has it simply come to mean "good," rather than "bad?" If it is shorthand for "friendship with the user" we might approach the topic by exploring what

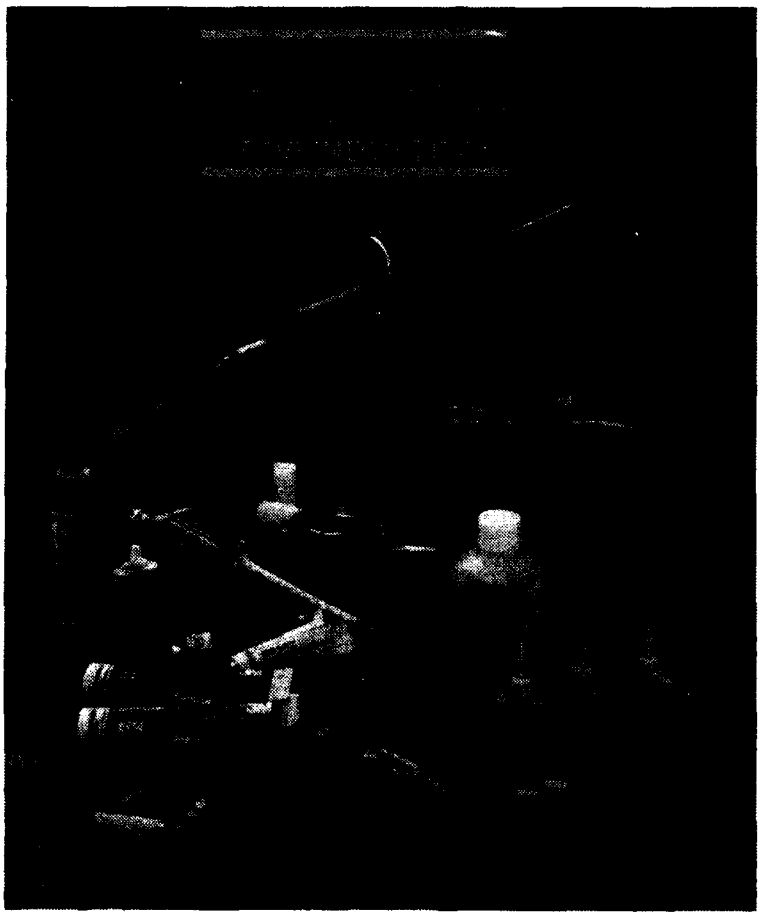

Fig. 1—Example usage of the term user-friendly

[1]This advertisement appeared in *Industrial Equipment News*, 1988. Cited without permission.

those words mean in this context: What kind of friendship—that of a salesman or that describing a long-standing mutual relationship among people? In the former case, it describes an easy, casual, assumed familiarity without depth or longevity. In the latter case, it involves a significant investment of time and energy to form an understanding of the resource that is the friend. Some software is like the salesman: We associate this friendliness with *easy-to-learn*; there is little investment in time and energy to get some utility, and the relationship might lack in depth and subtlety. Other software is like a deep friendship: We associate this friendliness with *easy-to-use*; a significant investment of time and energy pays back in a long-term relationship that is rewarding for its depth, dependability, and subtlety. (Continuing the analogy: Just as a good friend adapts to your needs, a good piece of software does too.)

Another approach to understanding "user-friendly" is to make finer distinctions in describing some of the attributes of the interface to computer programs, perhaps distinctions like that mentioned above: "easy-to-learn" vs. "easy-to-use." But are we merely replacing one phrase by two? No; we believe there are many "easy to . . ." dimensions, of which these two are only the most obvious. This Note follows this approach of making finer distinctions and attempts to explore a multi-dimensional space and describe its implications for understanding some of the attributes of a program's user-computer interface. Another interesting attempt to make finer distinctions is made by Jon Meads, in which he relates user-friendliness to systems having the attributes cooperative, preventative, conducive, reliable, predictable, and deferent (Meads, 1985).

Because they will help match software to users' needs, the distinctions addressed in this Note are important to persons who specify, design, develop, judge, select, develop training for, support, and use software systems and their associated user interfaces.

## II. THE DIMENSIONS OF "EASY TO . . ."

Software tools are meant to make our lives more productive and easier. But they do so in various ways, not all of which are mutually consistent or achievable. In fact, many of these ways tend to be mutually antagonistic. At present we distinguish the following ways a software's user interface might have the "easy to . . ." attribute. However, this is only one means of partitioning the space, based on our personal philosophies about software. Other categorizations may be more useful for other philosophies and their contexts. Our hope is to stimulate thought, not provide definitive categories.

### 1. EASY TO USE

Once a software package has been adequately learned—and without regard to how much time and effort that learning took—how easy is the program to use now? In short, can the learned user do what he or she wants to do easily?

### 2. EASY TO LEARN (AND TEACH)

This is often badly confused with "easy to use." We have seen this confusion occasionally evidenced by even writers and speakers of substantial reputation, knowledge, experience, and clarity of vision. How hard, measured in both time and intensity of effort, is it to acquire sufficient familiarity with the software's interface so that use becomes routine? This is quite distinct from easy-to-use: For example, user interfaces that lead you through each choice (e.g., through "drop down" menus ubiquitous in Macintosh and IBM's SAA architectures) might be easy to learn because of the "training wheels" they provide, but could get in the way of effective use.[1] The appropriateness of ease of learning is of course heavily dependent on the user's context. (We elaborate on context in a later section; see in particular our example of learning French for a trip to France as illustrating strong context-dependence.)

---

[1]To carry the analogy one step further: Some people question whether putting training wheels on bicycles is "just a crutch" creating the added complication of converting back to their nonuse. (This discontinuity is often mirrored in the differing form, vocabulary, structure, and content between "Getting Started" manuals and the reference manuals provided with software. Learning with one provides little help in moving to the other. The topic of software documentation is worthy of more extended consideration than can be given here.)

We note that this category is not at all the same as one we have deliberately avoided: "easy to demonstrate." Software that lends itself to quick, flashy though superficial demonstrations may well be very different from software that is easy to teach and to learn.

### 3. EASY TO RELEARN

The user has learned a software package, followed by a period of disuse. How hard is it to relearn the package? This is not the same as learning the first time, because it involves how well the models and ideas gained in the initial learning "stuck" in the user's cognition.

### 4. EASY TO UNLEARN

Edsger Dijkstra has stated that programmers who have learned BASIC become "mentally mutilated beyond hope of regeneration" (Dijkstra, 1982). Text editors are another example of systems that are often difficult to unlearn. Some systems might even be classed as addicting, in the sense that once the user learns them the user tends to stay with them. We believe that the concept of *inertia* applies, both to individuals and (even more) to organizations: To a degree rarely understood, learning the interface to a piece of software may entail making a lifetime commitment to it (or to software like it), at least in the sense that this act affects one's future attitude toward, and behavior with, other software systems. In organizations, inertia applies in spades: It may be very difficult to eradicate a piece of software once it is adopted.[2] We elaborate on this theme in describing "The Love/Hate Relationship," below.

### 5. EASY TO AVOID HARM

Is it easy to avoid causing inadvertent harm using the software interface? For example, through misunderstandings or misinterpretations at the user interface, user actions can cause harm by inadvertent alteration, destruction, or revelation of data (e.g., unintentionally mailing a reply to all those copied on the original message).

---

[2]A generalization of these notions gives rise to an important consideration: The *order* in which people acquire knowledge of programs is important and should be a topic of research and study.

### 6. EASY TO SUPPORT

Many organizations have evolved user support groups to aid their employees in the use of software packages; these groups often staff "hot lines" to be called for assistance and provide training classes. Similar user support is also furnished to customers by vendors of software. With the continuing increase in the number and diversity of software interface styles, ease of support has become critical. As the magnitude of support costs for a product becomes better understood, ease of support may well be a dominant factor in institutional purchases of software products.

* * * * *

We believe the above list of "easy to . . ." dimensions highlights the most important aspects of a system's user interface. But the list could easily go on. For example, user-computer interactions should be *easy to audit*, since it is often desirable, or even mandatory, to access a record of a system user's activities after the fact. (For example, a complete audit trail of a bank teller's use of his or her terminal is vital. An audit trail can be very helpful to user support, or even to users themselves in understanding "How did I get here?" or "Where am I?") Other factors deal less directly with the system interface and more with its functionality within an organizational context: A system should be *easy to share within a group*—that is, it should enhance computer-supported cooperative work. A system should be *easy to integrate into existing operations*, including (1) its integration with other hardware, software, and databases within an organization and (2) its integration with existing procedures (although they may evolve to new forms as a result of features in this and other software products).

Faced with a complex space of attributes like the above, an obvious reaction might be, "I want a user-computer software interface that meets the 'easy to' criteria along *all* dimensions." We believe, on the contrary, that you cannot have it all. Almost all of the dimensions tend to be mutually antagonistic; for example:

- If it is easy to learn and relearn, that may be because it is based on a clean, conceptually clear underlying model that can be assimilated so completely as to become virtually part of a user. However, that makes it harder to unlearn.

- Ease of use is often promoted by succinctness of operation; being able to issue a powerful command quickly, without pawing through four levels of menus. However, that succinctness and power can make it easier to do harm (e.g., to one's files) and requires a greater learning effort than verbose and overly descriptive program interfaces.
- If a program's interface provides quick access to significant power, with many useful capabilities, that may exacerbate the problems of creating, demonstrating, teaching, sharing, selling, and buying the product.

Not only are there antagonistic tradeoffs among the dimensions we have highlighted, the dimensions must also be traded against other important factors, not listed above because they are further removed from the "user friendly" issue. For example: Can the product be developed in a timely manner and sold for a reasonable price? Does the product fit into a well-understood niche, and does it interface with (some of) the industry standards? Is the product easy to maintain, and of course is the product functional? In fact, functionality is of extreme importance and can override all our proposed "easy to . . ." dimensions.

Given the mutual antagonisms, one must establish priorities, based on one's individual context. Therefore, we next discuss some attributes of a user's context that are relevant to establishing those priorities.

## III. THE CONTEXT: PEOPLE, TASKS

The dimensions of a software program's interface most important to a user depend—at a minimum—on attributes of the user and of the task being performed. Establishing priorities based on this context is difficult because that context is always changing. The very act of using a software program changes the user; it makes some other packages easier to learn (e.g., that share interface characteristics) and others harder to learn (e.g., because they are based on a conflicting model). Each acquisition and learning experience changes the equation.

However, there are unifying, stabilizing forces too. People are more similar than they are different. The user interfaces of two financial programs might be quite similar, although one is used by an engineer to keep track of household finances and the other by the corporate controller to aid in tracking institutional finances.

As individuals change, so does the social context. Entire societies are becoming more computer-literate and more familiar with certain industry standards (such as certain "look-and-feels"), thereby changing the entire market for which new products are designed. As we move into the 1990s, an ever larger share of the market will be occupied by relatively sophisticated and experienced but less adventurous users who have well-formed habits, likes, and dislikes.

We emphasize the multiple dimensions by which software interfaces should be judged and the richness of the context in determining relative priorities among those dimensions, because we often observe lack of attention to these factors in the design, procurement, and use of software packages. For example, we believe ease of learning often predominates when ease of use and support should; since these are often antagonistic qualities, ease of use then suffers Buyers of software, in becoming new users of it, overly focus on the learning task ahead of them, rather than the longer period of use that will follow. The syntactic richness, succinctness, and power that will provide eventual ease of use form higher obstacles to learning than the superficially friendly program that promises to be easy to learn.

Consider an analogy regarding these tradeoffs: If you need to learn some French for a forthcoming trip to France, your priorities depend greatly on the context. If you will spend two days there in first-class hotels and restaurants, a phrase book will probably do. If you are on a 4-month assignment and will be renting an apartment and shopping for food, a crash

Berlitz course is probably more appropriate. If you will be living there for several years, serious study of the language is in order.

People understand the tradeoffs sketched above and invest in learning appropriately based on expected time of learning vs. use. Why then do many (as we have conjectured) bias their priorities in favor of ease-of-learning or ease-of-buying in deciding on appropriate software for their task? Possibly due to relative unfamiliarity with the properties of software, so that many of the above-mentioned dimensions are wrongly assumed to be consonant, not antagonistic. Possibly due to the view of computers and software as a "universal tool," easily malleable into any needed configuration or attributes. In either case, we hope that discussion of the distinctions to be considered can aid in making appropriate tradeoffs in the acquisition and use of software.

## THE LOVE/HATE RELATIONSHIP

We believe study of the various dimensions of software characteristics might also aid in understanding a fascinating phenomenon related to software interfaces—there seem to be three distinct categories of software:

- *Software interfaces nobody loves.* This is often a property of the software itself, not of the person judging it. Everyone's favorite example in this category is IBM's Job Control Language (JCL).[1]
- *Software interfaces to which people are indifferent.* You get used to them, learning some makes others a bit harder to learn, but their utility is judged satisfactory. Examples in this category might be MS-DOS and the Pascal and C programming languages.
- *Software interfaces that are proselytizing.* These impart missionary zeal to many who come in contact with them. That is, users are not just content to use them, but insist that others use them too. Examples might include UNIX, the Star/Lisa/Macintosh-style interface, and the FORTH programming system.

Why does a software program generally fall into one of these three categories? We believe it is because of the interplay among all the factors cited in this Note: Software nobody loves tends to be deficient in all of the dimensions. Software that generates missionary zeal tends

[1]The reader who is not familiar with JCL need not worry about understanding this reference; just be thankful.

to be relatively skewed, emphasizing one or two of the dimensions at the expense of others. Interfaces in the middle category tend to be less skewed.

Study of the relationships among software interface traits, such as the set of dimensions we have discussed and the love/hate relationships the interfaces engender, can be useful. As descriptive categories for software, tasks, users, and organizations (and possibly other areas) are developed, we might find causal links predicting end users' reactions. In the meantime, corporate buyers of software might consciously consider how products they buy fit onto the hate/indifferent/addiction scale. For example, we can imagine deliberate decisions to avoid buying proselytizing software, so that its use and spread does not unacceptably bias future purchasing decisions. Further, developers of software might consciously design them to be proselytizing by better understanding the relationships among design dimensions, task, user, and organization that create this phenomenon.

## IV. CONCLUSION

We hope for a science of software interface design. Such a science would allow predictions of the effectiveness of a particular interface design given characteristics of the user, the task, the organizational context, and the interface design itself. The science must be based on prior development of useful distinctions, just as Lamarck was a necessary precursor to Darwin. To lay the foundation for such a science, we must now move beyond "user friendly" to distinctions in the design, procurement, and use of software systems that allow traits of the software to be prioritized and matched with traits of the user, task, and organization.

In developing a list of such user-computer interface dimensions, we are struck by: (1) how seldom some of these dimensions are explicitly considered in procurement and usage decisions; (2) the mutual antagonism of many of the dimensions; and (3) the inappropriate predominance of "easy-to-learn" over "easy-to-use" and "-support" in many design and procurement decisions. The antagonism among many of the dimensions leads to tradeoffs that must be considered. Better understanding of these tradeoffs should lead to better design and procurement decisions, leading in turn to the availability of more effective software products.

## REFERENCES

Anderson, R. H., N. Z. Shapiro, T. K. Bikson, and P. H. Kantar, *The Design of the MH Mail System*, The RAND Corporation, N-3017-IRIS, December 1989.

Dijkstra, Edsger W., "How Do We Tell Truths that Might Hurt?" *SIGPLAN Notices*, Vol. 17, No. 5, May 1982, pp. 13–15.

Eveland, J.D., and Tora K. Bikson, "Evolving Electronic Communication Networks: An Empirical Assessment," *Office: Technology and People*, Vol. 3, 1987, pp. 103–128.

Meads, Jon A., "Friendly or Frivolous," *DATAMATION*, Vol. 31, No. 7, April 1, 1985, pp. 96–100.

Shapiro, N. Z., and R. H. Anderson, *Toward an Ethics and Etiquette for Electronic Mail*, The RAND Corporation, R-3283-NSF/RC, July 1985.